User's Reference Manual

Commodore BASIC Version 4.0

commodore

Commodore BASIC Version 4.0

Appropriate for use with:
CBM and PET Computers

- Series 2001
- Series 3000
- Series 4000
- Series 8000

Part Number 321604

First Edition

July 1980

commodore

Introduction

Commodore BASIC 4.0 is the most sophisticated software developed for the CBM computer line to date. Programs are of course upward compatible from previous Commodore BASIC releases but both the user and the programmer can enjoy new features of 4.0

The first Commodore BASIC, version 2.0, was released in August 1977 for the PET 2001-8 computer. Version 3.0 in July 1978 added a machine language monitor and corrected known bugs of version 2.0. Version 3.0 is standard in all 2001 series since July 1978. Version 4.0, completed in October 1979 included all the improvements of previous releases, enhanced the speed of string processing, and integrated disk commands into the BASIC language.

BASIC 4.0 is standard in the CBM 4000 series and the CBM 8000 series. It is also available as an upgrade for CBM 2001 and 3000 series.

The information in this manual has been reviewed and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. The material in this manual is for information purposes only and is subject to change without notice.

Table of Contents

Chapter **1**

General Information About Commodore BASIC

1.1 INITIALIZATION

Cold start may be accomplished in any of three ways which all have the same resulting display as shown in Figure 1. The first, power-on, generates a system reset and initializes BASIC and OS variables. A reset signal may also be generated without power-on by a connection on the memory expansion connector. Finally, a SYS or branch to initialization code may be accomplished under program control.

```
*** commodore basic 4.0 ***

 31743 bytes free

ready.
```

Figure 1.

1.2 MODES OF OPERATION

In direct mode, BASIC commands and statements are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a “calculator” for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement [:BASIC statement] ... <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of S0 characters.

1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers control the execution of programs. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 63999.

1.4 CHARACTER SET

The Commodore BASIC character set is comprised of alphabetic characters, numeric characters, graphic characters, and special characters.

The following special characters and cursor characters are recognized by Commodore BASIC:

Character	Name
	Blank
	Semicolon
	Equal sign or assignment symbol +
Plus sign	
	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
f	Up arrow or exponentiation symbol
(	Left parenthesis
	Right parenthesis
	Percent
#	Number (or pound) sign
$	Dollar sign
!	Exclamation point
[	Left bracket
]	Right bracket
	Comma
	Period or decimal point
'	Single quotation mark (apostrophe)
"	Quotation mark
	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash
	At-sign
	Underscore

Character	Name
<DEL>	Deletes last character typed.
<carriage return>	Terminates input of a line.
<Shift><TAB>	Sets a tab stop at the current cursor position or clears tab if already set.
<TAB>	Move cursor to next TAB stop or end of line. Move cursor right one
<RIGHT>	column position. Move cursor left one column position. Clear screen and
<LEFT> <CLR>	home cursor.
<HOME>	Move cursor to upper left hand corner of screen.
<INST>	Move characters to right and including cursor position by one column to allow insertion of a single character.
<RVS>	Print all following characters until carriage return or <RVSOFF> in reverse video (black on white).
<RVSOFF>	Cease printing characters in reverse video.
<STOP>	Return control from program mode to direct mode (READY. and cursor is blinking.)
<RUN>	Inserts DLOAD"*",DO + <carriage return> as a command, loading and running the first program from the disk in drive 0.

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

"HELLO" "
525,000.00"
"Number of Employees"

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are two types of numeric constants:

1. Integer constants	Whole numbers between -32768 and +32767. Integer constants do not have decimal points.
2. Floating Point constants	Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The exponent must be in the range -38 to +37. Up to 9 significant digits can be represented. Examples: 235.988 E-4 = .0235988 2359 E6 = 2359000000

1.5.1 Binary Precision Form For Numeric Constants

In Commodore BASIC, all binary floating point constants are stored with 10 digits of precision, and printed with up to 9 digits.

A floating point binary constant is any numeric constant that has:

1. ten or fewer digits, or
2. exponential form using E, or
3. is assigned to a floating point binary type variable.

Examples of floating point binary constants:

46.8
- 7.09 E-06
2.5

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names and Declaration Characters

BASIC names may be any length, however, only the first two characters are significant. The characters allowed in a variable name are letters and numbers. The first character must be a letter. Special type declaration characters are also allowed-see below.

A variable name may not be a reserved word. The variable name cannot contain embedded reserved words. For example, if a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign (S) as the last character. For example:

AS = "SALES REPORT"

The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

Numeric variables may be declared to be integer variables by using a percent sign (%) following the variable name.

The default type for a numeric variable name is floating point binary.

Examples of BASIC variable names follow:

MI	declares a floating point binary variable
LIMIT%	declares an integer variable
NS	declares a string variable

1.6.2 Array Variables

An array is a group or table of values or strings referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with integers or integer expressions. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimensional array, T(1,4) would reference a value in a two-dimensional array, and so on. An array may contain a maximum of 32767 elements. An OUT OF MEMORY error occurs long before this with the standard 32k of memory.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "TYPE MISMATCH" error occurs.)

   Example:
   ```
   10 A%=23.42
   20 PRINT A%
   RUN
   23
   ```

2. All arithmetic and relational operations are performed in floating point binary. Integers are converted to floating point binary form for the evaluation of the expression, and then converted back into integers.
3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range - 32768 to 32767 or an "OVERFLOW" error occurs.
4. When a floating point value is converted to an integer, the fractional portion is truncated and the integer result is less than or equal to the floating point value.

   Example:
   ```
   10 C% = 55.88
   20 PRINT C%
   RUN
   55
   ```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC may be divided into four categories:

I. Arithmetic
2. Relational
3. Logical
4. Functional

1.8.1 Arithmetic Operators

[lie arithmetic operators, in order of preference, are:

Operator	Operation	Sample
T	Exponentiation	X|
	Negation	-X
*,J	Multiplication, Floating	X*Y
	Point Division	X/Y
+.-	Addition. Subtraction	X+Y
		X-Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Maximum nesting of parentheses for binary arithmetic is 10 levels.

Here are some sample algebraic expressions and their BASIC counterparts.

ALGEBRAIC EXPRESSION	BASIC EXPRESSION
X+2Y	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)Y$	(X|2)TY
XY^Z	XT(YTZ)
X(-Y)	X*(-Y) Two consecutive operators must be separated by parentheses.

1.8.1.1 Overflow and Division By Zero

If, during the evaluation of an expression, a division by zero is encountered, the "DIVISION BY ZERO" error message is displayed. If the evaluation of an exponentiation results in zero being raised to a negative power, the "DIVISION BY ZERO" error message is displayed.

If overflow occurs, the "OVERFLOW" error message is displayed.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Section 2.24)

OPERATOR	RELATION TESTED	EXPRESSION
	Equality	X=Y
O	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2 .27.)
When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression:

X+Y < (T-1)/Z

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples: IF

SIN(X) < **0 GOTO 1000**
IF I-INT(I/J) <> 0 THEN K=K+1

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

•	NOT
1	0
0	1

AND

•	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

•	Y	XORY
1	1	1
1	0	1
0	1	1
0	0	0

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.24). For example:

```
IF D<200 AND F<4 THEN 80 IF
1>10 OR K<O THEN 50 IF NOT
P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16	63 = binary 1 1 1 11 1 and 16 = binary 10000, so 63 AND 16=16
15 AND 14 = 14	15 = binary I 1 1 1 and 14 = binary 1 110, so 15 AND 14 = 14 (binary 1110)
-1 AND 8=8	-1 =binary 1111111111111111 and 8 =binary 1000, so -1 AND 8=8
4 OR 2 = 6	4 = binary 100 and 2 = binary 10, so 4 or 2 = 6 (binary 1 10)
10 OR 10= 10	10 = binary 1010, so 1010 OR 1010 = 1010 (10)
-1 OR -2= -1	-1 =binary 1 1 1 1 1 11 1 111 11 11 1 and -2 = binary 111 11 11 11 11 11 110, so -1 or-2 = -1. The bit complement of sixteen zeros is sixteen ones, which is the two's complement representation of -1.
NOT X=-(X+1)	The two's complement of any integer is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. Commodore BASIC has intrinsic functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's intrinsic functions are described in Chapter 3.

Commodore BASIC allows user defined functions that are written by the programmer. See DEF FN, Section 2.12.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW" + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" "AB"
"FILENAME" = "FILENAME"
"X$" > "X#"
"CL " > "CL"
"kg" < "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78" where B$ = "8/12/78"
```

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the DELETE key. Once a character(s) has been deleted, simply continue typing the line as desired.

To correct program lines for a program that is currently in memory, simply list the line. Use the cursor keys to edit the line on the screen and press <RETURN> to reenter the line.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.30.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC detects an error that causes program execution to terminate, an error message is printed. For a complete list of BASIC error messages, see Appendix B.

Chapter 2

BASIC Commands and Statements

All of the Commodore BASIC commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for an instruction. See below for format notation.

Versions: Lists the versions of COMMODORE BASIC in which the instruction is available.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction is used.

Example: Shows sample programs or program segments that demonstrate the use of the instruction.

Format :Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (<>) are to be supplied by the user.
3. Items in square brackets ([ ]) are optional.
4. All punctuation except angle brackets and square brackets (i.e.., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).
6. Items separated by a vertical bar (|) are mutually exclusive; choose one.

SECTION INDEX TO BUSINESS BASIC COMMANDS AND STATEMENTS

2.1 APPEND

Format: APPEND#<file number>,"<name>"[,D<x>] [ON U<y>]

Versions: *4.0*

Purpose: To write additional data to the end of a CBM disk sequential file.

Remarks: APPEND is like a DOPEN (Section 2.16) except it applies only to sequential files.
The CBM disk pointers are positioned beyond the current end of file.
Additional data may then be written and the file re-closed.
Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.
Unit defaults to 8, drive to 0.

Example: APPEND#1,"MASTER"
reopen master file on unit 8 drive 0 with logical file #1 for future PRINT#.

2.2 BACKUP

Format: BACKUP D<x> TO D<y> [ON U<z>j

Versions: *4.0*

Purpose: To duplicate the entire contents of a CBM disk on another diskette.

Remarks: User specified drive numbers must be 0 or 1. Drive numbers in the BACKUP command must be unique. Unit defaults to 8.

Example: BACKUP DO TO D1

2.3 CLOSE

Formal: CLOSE <file number>

Versions: *1.0,.0,3.0,4.0*

Purpose: *To* conclude I/O to a channel.

Remarks: <file number> is the number under which the file was OPENed.
The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any

111 c.

A CLOSE for a sequential output file writes the final buffer of output.

Example: See Appendix I.

2.4 CLR

Format *CLR*

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To set* all numeric variables to zero, all string variables to null, reset the end of memory and stack space, and free all array space.

Remarks: CLR may be executed by a BASIC program and that program can continue operation if the above conditions, particularly those affecting GOSUB, are observed.

Example: CLR

2.5 **CMD**

Format: CMD <file#>[,<print list>]

Versions: 1.0,2.0,3.0,4.0

Purpose: To address an IEEE bus device and leave it addressed and listening.

Remarks: CMD has the same parameter list as PRINT# (Section 2.34).

Example: *To* Create a Hardcopy Listing

```
OPEN 4,4
CMD4,"PROGRAM LISTING"
LIST
PRINT#4 (this turns off CMD)
CLOSE 4
```

2.6 COLLECT

Format: COLLECT [D<x>] [ON U<y>]

Versions: 4.0

Purpose: Frees up space allocated to improperly closed files on disk and deletes their references from the directory.

Remarks: An example of an improperly closed file is one that has been OPENed but never CLOSEd. Collect de-allocates the space they occupied and re-creates the bit allocation map by tracing through the file-links in the remaining files. Space allocated by the BLOCK-ALLOCATE command will be freed, since it is not linked to a file.

Example: COLLECT DO verifies drive 0 on unit 8
COLLECT verifies last drive accessed

2.7 CONCAT

Format: *CONCAT [D* <x>, *J*"<name l >"TO [D <y>,] "<name2>" [ON U <z>]

Versions: *4.0*

Purpose: Concatenate sequential files.

Remarks: Old file name 2 is deleted and replaced with a new file which is the concatenation of the two files. Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

Example: *CONCAT* "YOURFILE"TO"MYFILE"
MYFILE becomes MYFILE+YOURFILE

2.8 CONT

Format: CONT Versions:

1.0,2.0,3.0,4.0

Purpose. *To* continue program execution after the stop key has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (? or prompt string).
CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number.
CONT is invalid if the program has been edited during the break. Execution cannot be CONTinued if a direct mode error has occurred during the break.

Example: See example Section *2.43 STOP.*

2.9 ***COPY***

Format. *COPY* [D<x>,] "<name1>"TO [D<y>,] "<name2>"[ON U <z>]

Alternate
Format: *COPY* [D<x>] To [D<y>]

Versions: *4.0*

Purpose: Make a copy of a file within a disk unit.

Remarks: The copy command can only function between drives or a single drive within one unit. Copy without file names copies all files from Dx to Dy without altering files that already exist on Dy. Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

Example: *COPY* DO TO D1 copy all files on DO to D1
COPY D1,"TEXT" TO "BACKUP" create BACKUP on drive I

2.10 DATA

Format: DATA list of constants>

Versions: *1.0,2.0,3.0,4.0*

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section *2.46)*

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section *2.39)*.

2.11 DCLOSE

Format: DCLOSE[#<1>] [ON U <x>]

Versions: *4.0*

Purpose: Close disk files

Remarks: This command can close all files currently open on a disk unit or only the logical file specified. If no logical file number is specified, all files currently open will be closed. If both the logical file number and ON unit clauses are specified, the command still has the same effect as if only the logical file number was specified. In closing relative files, more records may be generated than were asked for, in order to fill out buffers in use.

Example: See also DOPEN (Section 2.16)

```
DCLOSE
DCLOSE#5
DCLOSE ON U7
```

2.12 **DEF FN**

Format: DEF FN<name> [(<parameter list>)] =<function definition>

Versions: *1.0,2.0,3.0,4.0*

Purpose: Define and name a function written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter> is comprised of the variable name in the function definition to be replaced when the function is called. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear as a parameter. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

User-defined string functions are not allowed. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "TYPE MISMATCH" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "UNDEFINED FUNCTION" error occurs. DEF FN is illegal in the direct mode.

Example:

```
410 DEF FNAB (X)=X 3/Y 2
420 T=FNAB (1)
```

Line *410* defines the function FNAB. The function is called in line 420.

2.13 l)IM

Format: DIM <list of subscripted variables>

Versions: 1.0,2.0,3.0,4.0

Purpose: *To* specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "BAD SUBSCRIPT" error occurs. The minimum value for a subscript is always 0.

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Matrices can have more than one dimension. Up to 255 dimensions are allowed but the size of each must be less than 32767, and the total array size is limited by the memory available.

Example: *10* DIM A(20)
20 FOR 1=0 TO 20
30 READ A(1) 40
NEXT 50 DATA 1,
2,3.. . **10 DIM R3(5,**
5) 10 DIM D$(2,2,2)

2.14 DIRECTORY

Format: DIRECTORY [D<x>] [ON U<y>]

Versions: *4.0*

Purpose: Display disk directory to screen.

Remarks: Displays both directories for the specified unit if no drive # is specified. A printer may be addressed by CMD so that this command can produce a hard-copy directory listing. Default unit is 8.

Pressing the STOP key terminates the listing.

On the 4000 series, pressing the SPACE bar causes listing to pause, pressing another key will restart the listing.

On the 8000 series, pressing the colon (:) causes the listing to pause, while 3, 6, or 9 on the top row will restart the listing. These keys will also work when shifted.

Example: DIRECTORY

DIRECTORY U8

OPEN4,4:CMD4: DIRECTORY: PRINT#4:CLOSE4

2.15 DLOA

Purpose: Load BASIC program file from d

Remarks: Default drive is 0, default unit is 8, and the name must be a string variable or a string enclosed in quotes.
Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.
DLOAD can be used in the body of a program to chain to other programs on the disk.

Example: DLOAD "MYFILE"
DLOAD "MYFILE", D1 ON U9

2.16 DOPEN

Format: DOPEN#<1>,"<name>" [,L<y>] [,D<x>] [ON U <z>] [,W]

Versions: *4.0*

Purpose: Declare a sequential or random access file for read or write.

Remarks: *#<I>. .logical* file number used to associate future disk operators back to this file open operation. l<=1<=255. Logical file numbers greater than 128 cause a carriage return and line feed to be sent with each PRINT#. Logical file numbers less than 128 send only the carriage return. The carriage return can be suppressed by the use of the semi-colon.
L<y>..record length causes allocation of a random access file with record length y: 1<=y<=255. Opened for read and write, by default.

D<x> drive number defaults to 0.

U<z> unit defaults to 8.
For sequential files, Write must be specified with a W, or the file will be opened to Read. Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

It is possible to replace an existing file with DOPEN, using the C?:

DOPEN#2,"@FILE1 ",D 1

Example: DOPEN#5,I AS) Where a$ ="filename"

2.17 DSAVE

Format: DSAVE "<name>" [,D<x>] [ON U<y>]

Versions: *4.0*

Purpose. Save a BASIC program the to disk.

Remarks: The file name can be specified up to 16 characters. Drive defaults to 0. Unit defaults to 8. When a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

It is possible to replace an existing file with @.

Example: DSAVE "PROGRAM" DSAVE
" 6r @PROGRAM".D1

2.18 END

Format: END

Versions: ***1.0,2.0,3.0,4.0***

Purpose: To terminate program execution, and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example: 520 IF K>1000 THEN END

OR Next

Format: FO

NEXT [<variable>] [,<variable>.
where x, y, and z are numeric expressions.

Versions: *1.0,2.0,3.0,4.0*

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement, on the next line. This is a FOR. . .NEXT loop. If STEP is specified as negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

NESTED LOOPS

FOR.. .NEXT loops may be nested, that is, a FOR. . .NEXT loop may be placed within the context of another FOR. . .NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. A single NEXT statement may be used for all of them, if it is followed by each variable, in order and separated by commas.

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT WITHOUT FOR" error message is issued and execution is terminated.

Due to stack limitations, maximum nesting is 9 FOR-NEXT loops.

Example 1: Nested Loops

```
10 FOR 1=1 TO 3
20 FOR J=1 TO 3
30 PRINT I;J
40 NEXTJ,I
RUN
 1 1 1
 2 1 3
 2 1 2
 2 2 3
 3 1 3
 2
```

Example 2: Variable Changes After Loop is Set *10*

```
K=10
20 FOR 1=1 TO K STEP 2
30 PRINT I;
40 K=K+10 50
PRINT K 60
NEXT RUN
  1     20
 3     30
 5 40
 7     50
 9     60
READY.
```

***Example** 3:* Second Value Less Than First

```
10 J=0
20 FOR 1=1 TO J
30 PRINT I 40
NEXT I
  RUN
   1
```

In this example, the loop will execute once because the initial value of the loop exceeds the final value, but is not tested until the NEXT is executed.

***Example** 4:* Variable Used Previously *10*

```
I=5 20 FOR 1=1 TO 1+5
30 PRINT l; 40 NEXT
RUN
  1          3 4 5 6
READY.
```

In this example, the loop executes six times. The initial value of the loop variable was set before the final value.

NOTE: You cannot use an integer variable (1%%) as the counter variable.

Example: *10* FOR I%=1 TO 10

```
20  PRINT  1%
30   NEXT   1%
RUN
?SYNTAX ERROR IN 10
```

2.20 GET

Format: GET[#<logical file number>,] <variable>

Versions: *1.0,2.0,3.0,4.0*

Purpose: To read a character from a file into a variable.

Remarks: GET without a logical file number scans the keyboard buffer and returns a value--numeric or string-if the buffer contains a keystroke. A null return for numeric variables is 0. A null return for string variables in LEN(A$)=0.

GET# reads one character from the logical file specified. If the device specified in the OPEN statement is 0 then GET# will act like GET. If it is 1 or 2 (cassette) then GET# will yield a carriage return on end-of-file condition, which can be detected by examining ST.

With timeout enables, a GET# from the IEEE sends a TALK, handshakes a byte, and sends an UNTALK. The variable ST will contain timeout and EOI data accordingly. If ST contains a timeout, then the variable will contain a carriage return.

Examples:

```
10PRINT "HANGS UNTIL KEY PRESSED"
20 GET AS: IF A$="" THEN 20
```

2.21 GOSUB... RETURN

Formal: *GOSUB line* number>

RETURN

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* branch to and return from a subroutine.

Remarks: line number> is the first line of the subroutine.
The RETURN statement(s) in a subroutine cause BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points of the subroutine. A subroutine may appear anywhere in the program, but it is distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine. An increase in execution speed can be observed by giving subroutines low numbers, preceding the body of the program. GOSUBs should be exited by RETURN. The use of GOTO to exit a subroutine will eventually result in an OUT OF MEMORY error as stack pointers build up. Maximum nesting is *23* GOSUBs.

Example:

```
10GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30END
40 PRINT "SUBROUTINE "; 50
PRINT "IN ";
60 PRINT "PROGRESS ";
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
READY.
```

2.22 GOTO

Format: GOTO <line number>

Versions: 1.0,2.0,3.0,4.0

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a non-executable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
10 READ R
20 PRINT "R = ";R,
30 A = 3.14*R|2
40 PRINT "AREA =";A
50 GOTO 10 60 DATA
5,7,12 RUN
R=5            AREA = 78.5
R=7            AREA= 153.86
R= 12          AREA=452.16 ?
OUT OF DATA ERROR IN 10
READY
```

2.23 HEADER

Format: HEADER "<disk name>", D<x> [,I<zz>] [ON U<y>]

Versions. *4.0*

Purpose: *To* format a blank disk or clear an old disk.

Remarks. When I<zz>, a disk II) number, is specified this command formats the disk specified. Otherwise the directory is cleared and the new name assigned to the disk. This command requires caution in its use.

A media error is possible with the HEADER command. This can be caused by a missing disk, write protect tab in place, or bad media. The HEADER command reads the disk command channel and if an error is encountered, will return the error message "?BAD DISK".

Whenever a variable or an evaluated expression is used as a disk name it must be surrounded by parentheses. The disk id (Izz) may not be specified with a variable.

Example: HEADER "MASTER DISK", DO, 101

2.24 IF...THEN and IF...GOTO

Format: If <expression> THEN <statement(s)> <line number>

Format: IF <expression> GOTO <Line number>

Versions: 1.0,2.0,3.0,4.0

Purpose. To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored. Execution continues with the next executable statement on the following line.
If...THEN statements may be nested. Nesting is limited only by the length of the line.

```
IF A=B THEN IF B=C THEN PRINT "A=C"
```

If an IF...THEN statement is followed by a line number in the direct mode, an " UNDEFINED STATEMENT" error results unless the resident BASIC program contains that line number.
NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS(A-1.0) <= 1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than I. OE-6.

Example 1: 200 IF I THEN GET I
This statement scans the keyboard for the number I if I is not zero.

***Example** 2:* 100 IF (1>10) AND (1<20) THEN DB=1979-1 :GOTO 300 110 PRINT "OUT OF RANGE"

2.25 INPUT

Format: INPUT ["<prompt string>";]<list of variables'-

Versions: 1.0.2.0.3.0.4.0

Purpose: To allow input from the keyboard during program execution.

Remarks: INPUT is illegal in the direct mode. When an INPUT statement is encountered, program execution pauses, a question mark is printed to indicate the program is waiting for data, and the cursor begins to blink. If <prompt string> is included, the string is printed before the question mark. The required data is then entered at the terminal.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

INPUT is limited to the length of a logical screen line. Because of the prompt, this is 78 characters. If more characters are entered, the INPUT will accept the last logical screen line.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. Strings input to an INPUT statement need not be surrounded by quotation marks.

Responding to INPUT with the wrong type of value (string instead of numeric, etc.) causes the message "?REDO FROM START" to be printed. No assignment of input values is made until an acceptable response is given. Responding to INPUT with too much data causes the message "?EXTRA IGNORED". Too little data causes an additional prompt ?? to *satisfy the missive* data.

Examples:

```
10 INPUT X
20 PRINT X "SQUARED IS" X 2
30 END
RUN
? 5 (The 5 was typed in by the user in response to            question mark.)
  5 SQUARED IS 25
READY.

LIST
10 PI=3.14159265
20 INPUT "WHAT IS THE RADIUS";R 30
A=PI*R|2
40 PRINT "THE AREA OF THE CIRCLE IS";A 50
PRINT
60 GOTO 20
READY.
RUN
WHAT IS RADIUS? 7.4 (User types 7.4) THE
AREA OF THE CIRCLE IS 172.033614

WHAT IS THE RADIUS?
etc.
```

2.26 INPUT#

Format: INPUT#<file number>,<variable list>

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* read items from a sequential or random disk file and assign them to program variables.

Remarks: <*file* number> is the number that was used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. The variable type must match the type specified by the variable name. INPUT# does not print a question mark as a prompt.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma. Instead of "?REDO FROM START" a "?FILE DATA ERROR" is reported. V-01 Or timeout forces an automatic carriage return on INPUT#.

INPUT# is limited to the maximum string size of 80 characters.

Examples: See Appendix I.

2.27 LET

Format: [LET] <variable>=<expression>

Versions: 1.0,2.0,3.0,4.0

Purpose: **To** assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12|2
130 LET F=12 |4
140 LET SUM=D+E+F
```

or

```
110 D=12
120 E=12|2
130 F=12|4
140 SUM=D+E+F
```

2.26 INPUT#

Format. INPUT#<file number>,<variable list>

Versions: *1.0,2.0,3.0,4.0*

Purpose: To read items from a sequential or random disk file and assign them to program variables.

Remarks: <file number> is the number that was used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. The variable type must match the type specified by the variable name. INPUT# does not print a question mark as a prompt.
The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. Leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma. Instead of "?REDO FROM START" a "?FILE DATA ERROR" is reported. EOI or timeout forces an automatic carriage return on INPUT#.

INPUT# is limited to the maximum string size of 80 characters.

Examples: See Appendix I.

2.27 LET

Format: [LET] <variable>=<expression>

Versions: 1.0,2.0,3.0,4.0

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=1212
130 LET F=1214
140 LET SUM=D+E+F
```

or

```
110 D=12
120 E=1212
130 F=1214
140 SUM=D+E+F
```

2.28 LIST

Format 1: LIST [<line number>]

***Format** 2:* LIST [<line number>] -[<line number>]

Versions: 1.0,2.0,3.0,4.0

Purpose: To list all or part of the program currently in memory to the active output device.

Remarks: BASIC always returns to command level after a LIST is executed.

***Format** 1:* If <line number> is omitted, the program is listed beginning at the lowest line number. Listing is terminated either by the end of the program or by typing STOP. If dine number> is included, only the specified line will be listed.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

On the 8000 series, pressing the colon (:) causes the listing to pause. Pressing 3, 6, or 9 on the top row restarts the listing. Any of these keys may be shifted.

***Examples: Format** 1:*

LIST	Lists the program currently in memory.
LIST 500	Lists line 500.

Format 2:

LIST 150-	Lists all lines from 150 to the end.
LIST -1000	Lists all lines from the lowest number through 1000.
LIST 150-1000	Lists lines 150 through 1000, inclusive.

2.29 LOAD

Format: LOAD "<filenanme>" [,<device #>1

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* load a file from external storage into memory.

Remarks: <filename> is the name that was used when the file was SAVEd.
LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. If LOAD is executed from one program, the loaded program is RUN after it is LOADed, and all open data files are kept open. Thus, LOAD may be used to chain several programs (or segments of the same program). None of the variables are cleared during a chain operation.
When the disk, device 8, is specified, the filename may be preceded by a drive number and a colon (see example). If no drive number is specified, both drives are searched.
Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

E=xample: LOAD "O:STRTRK",8

2.30 **NEW**

Format: NEW

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC always returns to command level after a NEW is executed.

2.31 ON. . .GOSUB and ON...GOTO

Format: ON <expression> GOTO <list of line numbers>
ON <expression> GOSUB <list of line numbers>

Versions: 1.0,2.0,3.0,4.0

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expressions> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded down.)

In the ON. . .GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is negative, an "ILLEGAL QUANTITY" error occurs. If the value is zero, or greater than the number of items in the list, control passes to the statement or line following.

Example: 100 ON L-l GOTO 150,300,320,390

2.32 **OPEN**

Format: OPEN <Logical file number> [,<device number> [,< secondary address> [,"<file name><parameters> 11]

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* establish an I/O channel over the IEEE bus or internal devices.

Remarks: Logical file number x must be specified such that *1<=x<=255.* Device number defaults to 1, the first cassette. Secondary address and file name default to none. On the IEEE the secondary address and the device number are sent with every GET#, INPUT#, and PRINT#.

Logical file numbers greater than *128* send a carriage return a line feed with each PRINT#. Logical file numbers less than *128* send only the carriage return. The carriage return can be suppressed with a semi-colon.

Type of file is Program unless S (Sequential) is specified. Sequential files are OPENed to READ unless W (Write) is specified. Drive number must be specified if the device addressed is a *2040* disk drive. For other models, drive number defaults to 0.

Files can be OPENed to tape (device I or *2),* disk (device *8),* printer (device *4),* or screen (device *3).*

Example: 10 OPEN 2,8,2,"0:DATAFILE,S,W" *20*
FOR 1=1 TO 10 *30* A$=CHR$(I) *40*
PRINT#1,A$ *50* NEXT I
60 CLOSE2

2.33 POKE

Format: POKE, I,J
Where I and J are integer expressions.
Versions: 1.0,2.0,3.0,4.0

Purpose: To write a byte into memory location.
Remarks: The integer expression I is the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255, 1 must be in the range 0 to 65535.
The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. (See PEEK Section 3.15.)
POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example: 10 POKE 59468,14
This sets the character set to business mode (upper/lower case).
20 POKE 59468,12
This resets the character set to graphic mode (upper case/graphic)

2.34 **PRINT and PRINT#**

Format: PRINT[#<logical file>,] [<list of expressions>]

Versions: *1.0,2.0,3.0,4.0*

Purpose: To output data to screen or specified channel.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. Strings must be enclosed in quotation marks.

PRINT POSITIONS: The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC divides the line into print zones of 10 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon. If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than *80* columns, the printing wraps to the next physical line.

Carriage return is also sent with each PRINT# statement. If the logical file number is greater than *128,* line feed is also sent. Both can be suppressed with a semicolon.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Any number x such that 0<x<.0I is output in exponential format.

A question mark may be used in place of the word PRINT in a PRINT *Statement.*

Example *1:*

```
10 X =5
20 PRINT X +5, X -5, X *(-5), X |5
30 END
RUN
  10         0         -25         3125
READY.
```

In this example, the commas in the PRINT statement cause each value to be at the beginning of the next print zone.

***Example** 2: 10* INPUT X

```
20 PRINT X "SQUARED IS" X|2 "AND";
30 PRINT X"CUBED IS" X|3 40 PRINT
50 GOTO 10
RUN
? 9
  9 SQUARED IS 81 AND 9 CUBED IS 729


? 21
  21 SQUARED IS 441 AND 21 CUBED IS 9261
```

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

***Example** 3: 10* FOR X = 1 TO 5

```
20 J=J+5 30 K=K+
10 40 ?J;K; 50 NEXT
X RUN
  5 10 10 20 15 30 20 40 25 50
READY.
```

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT for entry of the line. When line 40 is LISTed it will be a PRINT.

2.35 READ

Format: READ <list of variables>

Versions: 1.0,2.0,3.0,4.0

Purpose: *To* read data from a DATA statement and assign it to variables. (See DATA, Section 2.10.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a oneto-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "SYNTAX" error will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.39).

Example 1:

```
80 FOR I=1 TO 10
90 READ A(I) 100
NEXT
 110 DATA 3.08,5.19,3.12,3.98,4.24
 120 DATA 5.08,5.55,4.00,3.16,3.37
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2:

```
10 PRINT "CITY","STATE","ZIP" 20
READ C$,S$,Z
30 DATA "DENVER","COLORADO",80211 40
PRINT C$,S$,Z
 READY.
 RUN
 CITY          STATE          ZIP
DENVER COLORADO                80211
READY.
```

This program READs string and numeric data from DATA statement in line 30.

2.36 RECORD

Format: RECORD#<logical file>, <record> [,<byte>]

Versions: *4.0*

Purpose: *Used before (GET#,* INPUT# or PRINT# to position the record pointer in a random access file.

Remarks: Record number r such that O<r<=65535. Byte pointer b such that 1<=b<=254. Default value is 1.
When record pointer is set beyond last record the following conditions occur: If PRINT# is used, the appropriate number of records are generated to expand the file to the desired record. If INPUT# is used, a null line is returned with EOI in variable ST. If the record pointer is set beyond the end of the current end-of-file, DS$ will contain "RECORD NOT PRESENT".
Whenever a variable or an evaluated expression is used as a record number it must be surrounded by parentheses.
See Appendix I.

Example: *10* DOPEN#1,"RANDOM",L80 20
FOR 1=1 TO 10 30 RECORD#1,(
I) 40 PRINT#1,"ABCDEFG" 50
NEXT 60 DCLOSE#1

2.37 **REM**

Format: REM [<remark>]

Versions: *1.0,2.0,3.0,4.0*

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: **REM** statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

A remark may not be followed by another command on the same line, since the separating colon will be considered part of the remark.

Example:

```
120 REM CALCULATE AVERAGE VELOCITY
130 FOR 1=1 TO 20
140 SUM=SUM + V(I)
```

2.38 RENAME

Format: RENAME [D<x>,] "<old name>" TO "<new name" [ON U<y>]

Versions: 4.0

Purpose: Change the name of a disk file.

Remarks: The disk will not execute RENAME on any currently open file. Drive defaults to 0 and Unit defaults to 8.
Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

Example: RENAME "MASTER" TO "BACKUP"

2.39 RESTORE

Format: RESTORE

Versions: *1.0,2.0,3.0,4.0*

Purpose: To allow DATA statements to be reread from the beginning.

Remarks. After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program.

Example:

```
10 READ A,B,C,
20 RESTORE 30
READ D,E,F
40 DATA 57, 68, 79
```

2.40 RUN

Format: RUN [line number>]

Versions: *1.0,2.0,3.0,4.0*

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number.

A RUN stops and BASIC returns to command level when:

--there are no more line numbers to be executed -
-an END or STOP statement is executed a fatal
error occurs during execution.

2.41 ***SAVE***

Format: SAVE ["<filename>" [,<device number> [,<command>]]]

Versions: *1.0,2.0,3.0,4.0*

Purpose: *To* save a BASIC program file on tape or disk.

Remarks: Device number defaults to cassette number 1. Command can be 0-no end of tape written after the program is saved-or non-zero-an end of tape block is written. The file name defaults to null.

Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

If the disk (device 8) is specified, <filename> may be preceded by the drive number and a colon (see example). If no drive number is specified, the file is written on the last drive accessed.

Examples:

```
SAVE
SAVE "MYFILE"
SAVE "",2
SAVE "1:DISKFILE",8
```

2.42 SCRATCH

Format: Scratch "<name>" [,Dx] [ON U<y>]

Versions: *4.0*

Purpose: Scratch a disk file.

Remarks: "ARE YOU SURE?" prompt is given in direct mode to which the user must respond "yes"<return> or "y"<return>. Drive defaults to 0. Unit defaults to 8.
Whenever a variable or an evaluated expression is used as a filename it must be surrounded by parentheses.

Example: SCRATCH "MYFILE"
ARE YOU SURE? YES 01,
FILES SCRATCHED,01,00
READY.

2.43 STOP

Format: *STOP*

Versions: *1.0,2.0,3.0.4,0*

Purpose: *To* terminate program execution and return to command level.

Remarks: *STOP* statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed

BREAK IN LINE nnnnn

The STOP statement does not close files.
BASIC always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example:

```
10 INPUT A,B,C
20 K=(A+3)/2:L=B*3
30 STOP
40 M=C*K+100:PRINT M
RUN
'? 1,2,3
BREAK IN 30
READY
PRINT L
 6
READY
CONT
 106 READY
```

2.44 SYS

Format: SYS <variable name> [(<argument list>)]

Versions: 1.0,2.0,3.0,4.0

Purpose: To call an assembly language subroutine. (See also the USR function, Section 3.29).

<variable name>	contains an address that is the starting point in memory of a subroutine.
<variable name>	may not be an array variable name.
<argument list>	contains the arguments that can be passed to an assembly language subroutine.

Example:

```
110 MYROUT=30200 120
SYS MYROUT (A1)
```

2.45 VERIFY

Format: VERIFY ["<file name >- [,<device>]

Versions: *1.0, 2.0,3.0,4.0*

Purpose: Compare contents of program in memory to file on disk or tape and report differences.

Remarks: Device defaults to cassette #1. Name defaults to null. Name can only be null with tape devices.

Example: VERIFY"MYFILE" PRESS
PLAY ON TAPE 1 OK
VERIFYING
VERIFY ERROR
READY.

2.46 WAIT

Format. *WAIT* <address>, 1[,J]

Versions: *1.0,2.0,3.0,4.0*

Purpose: To suspend program execution while monitoring the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine address develops a specified bit pattern. The data read at the address is exclusive ORed with the integer expression J, and then ANDed with I. If the result is zero, *BASIC loops* back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero.
CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example: *100* WAIT 59411,8,8

This statement suspends execution until PLAY is pressed on the cassette unit.

Chapter 3

Commodore BASIC Functions

The intrinsic functions provided by BASIC are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expression
I and J	Represent integer expressions
X$ and Y$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC will truncate the fractional portion and use the resulting integer.

TABLE OF COMMODORE BASIC' FUNCTIONS

FUNCTION	RESULT		SECTION
	Numeric	String	
ABS	X		3.1
ASC	X		3.2
ATN	X		3.3
CHRS		X	3.4
COS	X		3.5
DS	X		3.6
DSS		X	3.7
EXP	X		3.8
ERE	X		3.9
INT	X		3.10
LEFT$		X	3.11
LEN	X		3.12
LOG	X		3.13
MIDS		X	3.14
PEEK	X		3.15
POS	X		3.16
RIGHTS		X	3.17
RND	X		3.18
SGN	X		3.19
SIN	X		3.20
SPC		X	3.21
SQR	X		3.22
STATUS	X		3.23
STRS		X	3.24
TAB		X	3.25
TAN	X		3.26
TIME	X		3.27
TIMES		X	3.28
USR	X		3.29
VAL	X		3.30

3.1 ABS

Format: *ABS(X)*

Versions: *1.0,2.0,3.0,4.0*

Action: Returns the absolute value of the expression X.

Example: **PRINT ABS(7*(-5))** *35*
READY.

3.2 ASC

Format: ASC(X$)

Versions: *1.0,2.0,3.0,4.0*

Action: Returns a numerical value that is the ASCII code of the first character of the string X$. (See Appendix D for ASCII codes.) If X$ is null, an "ILLEGAL QUANTITY" error is returned.

Example:
```
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
READY.
```

See the CHRS function for ASCII-to-string conversion.

Versions: 1.0,2.0,3.0,4

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. 7 expression X may be any numeric type, but the evaluation of ATN is always performed in floating point binary.

Example:
```
10 INPUT X
20 PRINT ATN(X)
RUN
?3
  1.24904577
READY.
```

3.4 CHRS

format: **CHR$(I)**

Versions: **1.0,2.0,3.0,4.0**

Action: Returns a string whose one element has ASCII code l. (ASCII codes are listed in Appendix D.) CI IRS is commonly used to send a special character to the terminal. For instance, a screen clear could be sent (CIIR$(147)) to clear the CRT screen and return the cursor to the home position, as a preface to an error message.

Example: PRINT CHR$(66)

B

READY.

See the ASC function (Section 3.2) for ASCII-to-numeric conversion.

3.5 COS

Format: *COS(X)*

Versions: *1.0,2.0,3.0,4.0*

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in floating point binary.

Example:

```
10 X=2*COS(.4)
20 PRINT X
RUN
 1.84212199
READY.
```

3.6 DS

Format: *DS*

Versions: 4.0

Action: Returns the error code corresponding, to the last CBM disk operation. See CBM DOS manual for error codes and definitions.

If no device responds to DS, the value returned is 0<=DS<20.

Example: *10* DOPEN#2,"MASTER FILE"
20 IF DS >= 20 THEN PRINT "ERROR":STOP

3.7 DSS

Format: *DS$*

Versions: *4.0*

action: Returns the disk status string corresponding to the last operation on a CBM disk. The string consists of an error code, message, track and sector. Fields are separated by commas. See CBM DOS manual for error codes, messages and definitions.

If the disk device is not present, LEN(DS$)=0.

Example: PRINT DS$

00,OK,00,00

3.8 EXP

Format: *EXP(X)*

Versions: 1.0,2.0,3.0,4.0

Action: Returns e to the power of X. X must be <= 88.0296919 1. If EXP overflows, the "OVERFLOW' error message is displayed.

Example:

```
10 X=5
20 PRINT EXP (X-1)
RUN
 54.5981501
READY.
```

3.9 FRE

Format: FRE(X)

Versions: *1.0,2.0,3.0,4.0*

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC. The FRE function forces a string garbage collection.

Example: PRINT FRE(0)
14542 READY.

3.10 INT

Format: *INT(X)*

Versions: *1.0,2.0,3.0,4.0*

Action: Returns the largest integer <=X.

Examples: PRINT INT(99.89) 99
READY.

PRINT INT(-12.11)
-13
READY.

3.11 LEFT$

Format: LEFT$

Versions: *1-0.2[1].0.3.0.4.0*

Action: Returns a string comprised of the leftmost I characters of X$. I must be in t range 0 to 255. If I is greater than LEN(X$), the entire string (X$) will be re turned. If 1=0, the null string (length zero) is returned.

Example: 10 A$ = "COMMODORE BASIC" 20 B$ = LEFT$(A$,9) 30 PRINT B$
COMMODORE READY.

Also see the MID$ and RIGHTS functions.

3.12 LEN

Format: LEN(X$)

Versions: 1.0,2.0,3.0,4.0

Action: Returns the number of characters in X$. Non-printing characters and blanks are counted.

Example: *10 X$* = "SANTA CLARA, CALIFORNIA" 20
PRINT LEN(X$) 23
READY.

3.13 LOG

Format: LOG(X)

Versions: *1.0,2.0,3.0,4.0*

Action: Returns the natural logarithm of X. X must be greater than zero.

Example: PRINT *LOG(45/7) 1.*
86075234 READY.

3.14 MID$

Format: MID$(X$,I[,J])

Versions: *1.0,2.0,3.0,4.0*

Action: Returns a string of length J characters from XS beginning with the Ith character. I and J must be in the range 0 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X$), MID$ returns a null string.

Example: LIST

```
10 A$="GOOD"
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
READY.
```

Also see the LEFT$ and RIGHT$ functions

3.15 PEEK

Format: PEEK(I) Versions:

1.0,2.0,3.0,4.0

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location 1. I must be in the range 0 to 65535. PEEK is the complementary function to the POKE statement.
In version 2.0 (the original 8k PET) if you PEEK a BASIC area a value of zero is returned.

Example: A=PEEK(59468)

3.16 POS

Format: *POS(X)*

Versions: 1.0,2.0,3.0,4.0

Action: Returns the current cursor position. The leftmost position is 0. X is a dummy argument.

Example: IF POS(X)>60 THEN PRINT CHR$(13)

3.17 RIGHTS

Format: RI(GHT$(X$,I)

Versions: *1.0,2.0,3.0,4.0*

I

Action: Returns the rightmost I characters of string X$. If I=LEN(X$), returns X$. If 1= 0, the null string (length zero) is returned.

Example: *10* A$="COMMODORE BASIC" 20 PRINT RIGHT$(A$,5) RUN
BASIC
READY.

Also see the MID$ and LEFT$ functions.

3.18 RND

Format: RND(X)

Versions: 1.0,2.0,3.0,4.0

Action: Returns a random number between 0 and 1. X>0 returns the same pseudo-random number sequence for any random number seed. X<0 re-seeds the sequence, each x producing a different seed. The sequence is seeded at random on power-up.
X=0 generates a random number from a free running clock. This feature is not available in version 2.0, original 8k PET.

Example:

```
10 FOR 1=1 TO 5
20 PRINT INT(RND(X)* 100);
30 NEXT
RUN
  24 30  31  51 5
READY.
```

3.19 SGN

Format: SGN(X)

Versions: 1.0,2.0,3.0,4.0

Action: IF X>0, SGN(X) returns 1. IF X= O, SGN(X) returns 0. IF X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

3.19 SGN

Format: SGN(X)

Versions: *1.0,2.0,3.0,4.0*

Action: IF X>0, SGN(X) returns 1. IF OX, SGN(X) returns 0. IF X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO *100,200,300* branches to 100 if X is is negative, 200 if X 0 and *300* if X is positive.

3.20 SIN

Format: SIN(X)

Versions: 1.0,2.0,3.0,4.0

Action: Returns the sine of X in radians. SIN(X) is calculated in floating point binary.
COS(X)=SIN(X+3.14159265/2)

Example: PRINT SIN(1.5) .
997494987
READY.

3.21 SPC

Format: *SPC(I)*

Versions: *1.0,2.0,3.0,4.0*

Action: Print I blanks on the terminal, starting at the cursor position. SPC may only be used with PRINT. I must be in the range 0 to 255.

Example: PRINT "OVER" SPC(15) "THERE"

```
OVER               THERE:
READY.
```

3.22 SQR

Format: SQR(X)

Versions: 1.0,2.0,3.0,4.0

Action: Returns the square root of X. X must be >=0.

Example:

```
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10                       3.16227766
 15                       3.87298335
 20                       4.47213595
 25                       5
READY.
```

3.23 STATUS

Format: STATUS

Versions: 1.0,2.0,3.0,4.0

Action: Returns the CBM status corresponding to the last I/O operation, whether over cassette, screen, keyboard or IEEE.

ST **bit position**	**ST** **numeric value**	**Cassette** **Read**	**IEEE** **R/W**	**Tape** verify + lo ml
0	1		time out write	
1	2		time out read	
2	4	short block		short block
3	8	long block		long block
4	16	unrecoverable read error		any mismatch
5	32	checksum error		checksum e r
6	64	end of file	EOI	
7	-128	end of tape	device not present	end of tape

Example:

```
10 DOPEN#2, "MASTER FILE" 12
GET#2,A$
14 IF STATUS AND 64 THEN 20
16 ?A$
18 GOTO 12
20 ?A$: DCLOSE#2
```

3.24 STRS

Format: *STR$(X)*

Versions: *1.0,2.0,3.0,4.0*

Action: Returns a string representation of value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
```

Also see the VAL function.

3.25 TAB

Format: TAB(I)

Versions: 1.0,2.0,3.0,4.0

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 0 is the left most position, and the rightmost position is the width minus one. I must be in the range 0 to 255. TAB may only be used in PRINT.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ AS, B$
30 PRINT A$ TAB(25)B$
40 DATA "G.T. JONES","$25.00"
RUN
NAME                    AMOUNT

G.T. JONES              $25.00
READY
```

3.26 TAN

Format: TAN(X)

Versions: 1.0,2.0,3.0,4.0

Action: Returns the tangent of X in radians. TAN(X) is calculated in binary. If TAN overflows, the "OVERFLOW" error message is displayed.

Example:

```
10 X = .785398163
20 Y = TAN(X)
30 PRINT Y
RUN
  1
```

3.27 TIME

Format: *TI*

Versions: *1.0,2.0,3.0,4.0*

Action: Used to read the internal clock and return a value in 1 /60ths of a second (jiffies).

Example: 10 ON TI GOTO 100,200,300

3.28 TIME$

Format: *TI$*

Versions: *1.0,2.0,3.0,4.0*

Action: Used to read the internal clock and return a string of 6 characters in hours, minutes, seconds. May be used in an input statement or on the left hand side of an expression to set the time of day.

Example:
```
10 TI$="000000"
20 FOR 1=1 TO 10000:
NEXT 30 PRINT TI$
RUN
000011
READY.
```

3.29 USR

Format: USR(X)

Versions: 1.0,2.0,3.0,4.0

Action: *Calls* the user's assembly language subroutine with the argument X. See Appendix for example.

Example: 40 B = T*SIN(Y) 50
C = USR(B/2) 60
D = USR(B/3)

3.30 VAL

Format: VAL(X$)

Versions: *1.0,2.0,3.0,4.0*

Action: Returns the numerical value of string X$. If the first character of X$ is not +, -, $, or a digit, VAL(X$)=0.

Example:

```
10 READ NAME$, CITY$, STATE$, ZIP$
20 IF VAL(ZIP$)<9000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL (ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
```

See the STRS function for numeric to string conversion.

Appendix

A

Commodore BASIC Compatibility

A. 1 COMMODORE BASIC UPWARD COMPATIBILITY

All versions of Commodore BASIC are upward compatible with regard to structure and statement syntax. To conserve user space, BASIC keywords are translated to 1 character tokens. Versions l-4 have maintained the same values for tokens of keywords common between the versions. The token values are shown by the following table.

TOKEN TABLE FOR COMMODORE BASIC

TOKEN	VERSION l-3 KEYWORD	VERSION 4 KEYWORD
128	END	
129	FOR	
130	NEXT	
131	DATA	
132	INPUT#	
133	INPUT	
134	DIM	
135	READ	
136	LET	
137	GOTO	
138	RUN	
139	IF	
140	RESTORE	
141	GOSUB	
142	RETURN	
143	REM	
144	STOP	
145	ON	
146	WAIT	
147	LOAD	
148	SAVE	
149	VERIFY	

TOKEN TABLE FOR COMMODORE BASIC (cont.)

TOKEN	**VERSION** 1-3 KEYWORD	VERSION 4 KEYWORD
150	DEF	
151	POKE	
152	PRINT#	
153	PRINT	
154	CONT	
155	LIST	
156	CLR	
157	CMD	
158	SYS	
159	OPEN	
160	CLOSE	
161	GET	
162	NEW	
163	TAB	
164	TO	
165	F N	
166	SPC	
167	THEN	
168	NOT	
169	STEP	
170	+	
171	-	
172	+	
173	/	
174	t	
175	AND	
176	OR	
177	>	
178	=	
179	<	
180	SGN	
181	INT	
182	ABS	
183	USR	
184	FRE	
185	POS	
186	SQR	
187	RND	
188	LOG	
189	EXP	
190	COS	
191	SIN	
192	TAN	
193	ATN	
194	PEEK	
195	LEN	
196	STRS	
197	VAL	
198	ASC	
199	CHRS	
200	LEFT$	
201	RIGHTS	
202	MIDS	
203	GO	
204	(1) ?SYNTAX ERROR	CONCAT
205		DOPEN
206		DCLOSE
207		RECORD

TOKEN	VERSION 1-3 KEYWORD	VERSION 4 KEYWORD
208		HEADER
209		**COLLECT**
210		BACKUP
211		COPY
212		APPEND
213		DSAVE
215		CATALOG
216		RENAME
217		SCRATCH
218		DIRECTORY
219		(1) ?SYNTAX ERROR
Note (1): Further tokens produce this error.		

A.2 CONVERTING OTHER PROGRAMS TO COMMODORE BASIC

If you have programs written in a BASIC other than Commodore BASIC, some minor adjustments may be necessary before running them with Commodore BASIC. Here are some specific things to look for when converting BASIC programs.

\.2.1 String Dimensions

Delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the Commodore BASIC statement DIM A$(J).

Some BASICS use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for Commodore BASIC string concatenation.

In Commodore BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	Commodore BASIC
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

A.2.2 Multiple Assignments

Some BASIC's allow statements of the form:

```
10 LET B=C=O
```

to set B and C equal to zero. Commodore BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=O: B=0
```

A.2.3 **Multiple Statements**

Some BASICs use a backslash (\) to separate multiple statements on a line. With Commodore BASIC, be sure all statements on a line are separated by a colon (:).

A.2.4 MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR.. .NEXT loops to execute properly.

Appendix B

Summary of Error Messages

BASIC ERRORS	OS ERRORS
BAD SUBSCRIPT CAN'T CONTINUE DIVISION BY ZERO FILE DATA FORMULA TOO COMPLEX ILLEGAL DIRECT ILLEGAL QUANTITY NEXT WITHOUT FOR OUT OF DATA OUT OF MEMORY OVERFLOW REDIM'D ARRAY REDO FROM START RETURN WITHOUT GOSUB STRING TOO LONG SYNTAX TYPE MISMATCH UNDEF'D FUNCTION UNDEF'D STATEMENT	BAD DATA BAD DISK DEVICE NOT PRESENT FILE NOT FOUND FILE NOT OPEN FILE OPEN LOAD NOT INPUT FILE NOT OUTPUT FILE VERIFY

Resumption of execution is not permitted with a CONT command. Variables within the statement or program retain their values so they may be scrutinized to determine a cause of error, if necessary. GOSUB and FOR entries on the stack at the time of error are cleared so resumption of execution is not possible by RETURN or NEXT.

Possible BASIC Messages and Meanings

BAD SUBSCRIPT-An attempt was made to reference a matrix element which is outside the dimensions of the matrix. This may happen by specifying the wrong number of dimensions or a subscript larger than specified in the original dimension.

```
DIM A(2,2)
A(1,1,1)=2
?BAD SUBSCRIPT ERROR
READY.
A(10,10)=2
?BAD SUBSCRIPT ERROR
READY.
```

CAN'T CONTINUE-Program execution cannot be resumed via a CONT command in four cases:

1) no program exists.
2) a new line was just typed in.
3) the program has not recently been run. 4) an error just occurred.

```
10A$="HELLO"
CONT
?CAN'T CONTINUE ERROR
READY.
```

DIVISION BY ZERO- Zero as a divisor would result in numeric overflow-thus it is not allowed. When this message appears, it is most expedient to list the statement and look for division operators.

```
?DIVISION BY ZERO ERROR IN 10
LIST 10
10 A=B/C
?UNDEF'D FUNCTION ERROR
READY.
```

FILE DATA--Occurs when an INPUT# statement finds a string while attempting to read a numeric value.

FORMULA TOO COMPLEX-This indicates that BASIC has run out of string temporary pointers to keep track of substrings in evaluating a string expression.

?FORMULA TOO COMPLEX ERROR
READY.

Break the string expression into two smaller parts to cure the problem.

ILLEGAL DIRECT -A single 80 column buffer area is used by BASIC to process incoming characters. This same buffer is used to hold a statement that is being interpreted in direct mode. INPUT will not work because incoming characters would overwrite the variable list following INPUT to be processed.

DEF cannot be used in direct mode for a different but similar reason. The name of a function is stored in the BASIC variable area with pointers to the string of characters which define the function. Since the function exists only in the input buffer, it would be wiped out the first time a new command is typed in.

INPUT A
?ILLEGAL DIRECT ERROR
READY.

ILLEGAL QUANTITY--Occurs when a function is accessed with a parameter out of range caused by:

I. A matrix subscript out of range (0 < X < 32767) X(-1)=Y
?ILLEGAL QUANTITY ERROR

2. LOG (negative or zero argument)

3. SQR (negative argument)

4. A B where A=0 and B not integer.
?(-5) 2.3 is illegal because it would give a complex result.

5. Call of USR before machine language subroutine has been patched in.

6. Use of string functions MID$, LEFT$, RIGHTS, with length parameters out of range (I <X<255).

7. Index on ...GOTO out of range.

8. Addresses specified for PEEK, POKE, WAIT, and SYS out of range (O<X<65535).

9. Byte parameters of WAIT, POKE, TAB and SPC out of range. (0<X<255).

POKE 32768,1000
?ILLEGAL QUANTITY ERROR
READY.

NEXT WITHOUT FOR-Either a NEXT is improperly nested or the variable in a NEXT statement corresponds to no previously executed FOR statement.

```
FOR 1=1 TO 10: NEXT:NEXT ?
NEXT WITHOUT FOR ERROR
READY.

FOR 1=1 TO 10: NEXT J
?NEXT WITHOUT FOR ERROR
READY.
```

OUT 01: DATA-A READ statement was executed but all of the data statements in the pro gram have been read. The program tried to read too much data, or insufficient data was included in the program. Carriage returning through a line READY on the CBM TV display, sometimes yields this error because the message is interpreted as READ Y.

```
READY.
?OUT OF DATA ERROR
READY.
```

OUT OF MEMORY--May appear while entering or editing a program as the text completely fills memory. At run time, assignment and creation of variables may also fill all variable memory. Array available declarations consume large areas of memory even though a program may be rather short. The maximum number of FOR loops and simultaneous GOSUBs are de-pendent on each other. This context is stored on the 6502 hardware stack whose capacity may be exceeded. To determine the type of memory error, print FRE(O). If there are a large number of bytes available, it is most likely a FOR-NEXT or GOSUB problem.

```
10GOSUB1
0
RUN
?OUT OF MEMORY ERROR IN 10
READY.
?FRE(
0)
31732
```

A subroutine which terminates in GOTO rather than RETURN will eventually cause an OUT OF MEMORY error as stack pointers build up.

OVERFLOW Numbers resulting from computations or input that are larger than binary 1. 70141 184 E+38 cannot be represented in BASIC's number format. Underflow is not a detectable error but numbers less than binary 2.93873587 E-39 are indistinguishable from zero.

```
?1E40
?OVERFLOW ERROR
READY.
```

REDIM'D ARRAY-After an array was dimensioned, another dimension statement for the same array was encountered. For example, an array variable is defined by default when it is first used, and later a DIM statement is encountered.

```
A(5)=6
DIM A(10,10)
?REDIM'D ARRAY ERROR
READY.
```

REDO FROM START-is not actually a fatal error printed in the standard format but is a diagnostic printed when data in response to INPUT is alpha when a numeric quantity is required.

```
10 INPUT A
RUN ?
ABC
?REDO FROM START
```

The INPUT continues to function until acceptable data has been received.

```
10 INPUT A,B,C
RUN
?1 ??2 ?
?3
READY.
```

RETURN WITHOUT GOSUB-A RETURN statement was encountered without a previous GOSUB statement being executed.

```
CLR
RETURN
?RETURN WITHOUT GOSUB ERROR
```

STRING TOO LONG Attempt by use of the concatenation operator to create a string more than 255 characters long.

```
A = A
FOR 1=1 TO 10:A$=A$+A$:NEXT ?
STRING TOO LONG ERROR READY.
```

SYNTAX BASIC cannot recognize the statement you have typed. Caused by such things as missing parenthesis, illegal characters, incorrect punctuation, misspelled keyword.

```
RUIN
?SYNTAX ERROR
READY.
```

TYPE MISMATCH-The left-handed side of an assignment statement was a numeric variable and the right-hand side was a string, or vice versa; or a function which expected a string argument was given a numeric one, or vice versa.

```
A$=5
?TYPE MISMATCH ERROR
 READY.
```

UNDEF'D STATEMENT An attempt was made to GOTO,GOSUB, or THEN to a statement which does not exist.

```
GOTO A
?UNDEF'D STATEMENT ERROR
 READY.
```

UNDEF'D FUNCTION Reference was made to a user defined function which had never been defined.

```
X=FNA(3)
?UNDEF'D FUNCTION ERROR
 READY.
```

B)

Operating System Messages and Meanings

BAD DATA Numeric data was expected but alpha data was received when inputing from a special device.

BAD DISK Media failure on a HEADER command.

DEVICE NOT PRESENT-No device on the IEEE was present to handshake an attention sequence. Status will have a value of 2 which corresponds to a time out. May happen on OPEN, CLOSE, CMD, INPUT#, GET#, PRINT#. If a filename is not specified with OPEN, this error will not occur.

```
OPEN 5,4,3, "FILE"
?DEVICE NOT PRESENT ERROR
READY.
```

FILE NOT FOUND-The named file specified in OPEN or LOAD was not found on the device specified. In the case of tape I/O, an end of tape mark was encountered.

FILE NOT OPEN-- The operating system must have device number and command information provided by the OPEN statement. If an attempt is made to read or write a file without having done this previously, then this message appears:

```
CLR
INPUT#10,A
?FILE NOT OPEN ERROR
READY.
```

FILE OPEN-An attempt to redefine file parameter information by repeating an OPEN command on the same file twice.

```
OPEN 1,4,1
OPEN 1,4,1
?FILE OPEN ERROR
READY.
```

LOAD Only occurs when loading a program from cassette tape. This means that there were more than 3 1 errors in the first tape block or that there were errors in exactly the same corresponding positions of both blocks.

NOT INPUT FILL--Tape files, once opened for writing, cannot be read without first CLOSEing, rewinding tape and OPENing for INPUT. This message appears when an attempt is made to read an output file:

```
10 OPEN 1,1,1
20 INPUT =+1,A
?NOT INPUT FILE ERROR
READY.
```

NOT OUTPUT Fl LL-Tape tiles cannot he read and updated in place. Device 0 is the keyhoard and it cannot he written to:

```
10 OPEN 1,0
20 PRINT -t1
?NOT OUTPUT FILE ERROR
READY.
```

VERIFY -The contents of memory and a specified file do not compare.

Appendix C

Mathematical Functions

DERIVED FUNCTIONS

Functions that are not intrinsic to Commodore BASIC may be calculated as follows:

FUNCTION	PET BASIC EQUIVALENT
SECANT	SEC(X)=I/COS(X)
COSECANT	CSC(X)=I/SIN(X)
COTANGENT	COT(X)=l/TAN(X)
INVERSE SINE	ARCSIN(X)=ATN(X/SQR(-X*X+1)
INVERSE COSINE	ARCCOS(X)=-ATN (X/SQR(-X*X+1))+pi/2
INVERSE SECANT	ARCS EC(X)=ATN(X/SQR(X*X-1))
INVERSE COSECANT	ARCCSC(X)=ATN(X/SQR(X*X-1)) +(SGN(X)-1)*pi/2
INVERSE COTANGENT	ARCOT(X)=ATN(X)+pi/2
HYPERBOLIC SINE	SINE(X)=(EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X)=(EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	'FAN II(X)=EXP(-X)/EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X)=2/(EXP(X)+EXP(-X)
HYPERBOLIC COSECANT	CSCH(X)=2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X)=FXP(-X)/(EXP(X)-EXP(-X)*2+1
INVERSE HYPERBOLIC SINE	ARCSINIi(X)=LOG(X+SQR(X*X+1)
INVERSE HYPERBOLIC COSINE	ARCCOSH(X)=LOG(X+SQR(X*X-1)
INVERSE HYPERBOLIC TANGENT	ARCTANH(X)=LOG((l+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)/X
INVERSE HYPERBOLIC COTANGENT	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Appendix D

ASCII Character Codes

ASCII CODE	CHARACTER	ASCII CODE	CHARACTER	ASCII CODE	CHARACTER
000	NULL	043	+	086	V
001	SOH	044		087	w
002	**STX**	045	-	088	X
003	ETX	046		089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	1
006	ACK	049	1	092	bkslash
007	BEL	050	2	093	1
008	BS	051	3	094	t
009	HT	052	4	095	back arr
010	LF	053	5	096	space
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058		101	e
016	DLE	059		102	f
017	DCl	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	(u	107	k
022	SYN	065	A	108	1
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	1	116	t
031	US	074	1	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034		077	M	120	x
035	#	078	N	121	y
036	$	079	0	122	z
037	%	080	P	123	
038	&	081	Q	124	<
039		082	R	125	=
040		083	S	126	>
041		084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal

Appendix E

CBM Display Character Codes

DEFINITION OF TERMS

CBM Program Code-Character codes as they are stored in BASIC e.g., PRINT statements. Very close to standard ASCII.

CBM Display Code-Character codes for TV display circuitry. Basically uses 6 bits to describe 64 characters, Bit 6 for alternate character set, and Bit 7 for reverse field.

Conversion Between Codes

Display code- -A   Program code-- - -B- - - ASCII

It is possible to write simple logic formulas to convert between all three codes for the 64 ASCII upper case symbols and the 32 ASCII lower case symbols. The only symbols which have no ASCII symbol equivalent are the graphics characters.
Because there are two character generator versions for CBM and two character sets in each generator, the user will have to decide which pair of formulas will be required to accomplish the conversions shown in the figure above.

1. To translate to standard ASCII from CBM program code given character code X:

```
IF $41<=X<=$5A THEN X=X.OR.$20

ELSE IF $C1<=X<=$DA THEN X=X.AND.$7F
```

Now all characters X > $7B are CBM characters with no ASCII equivalent.

2. To translate to standard ASCII from Graphics CBM program code:

No translation needed for character codes <_ $5F.
All other characters have no ASCII equivalent.

3. To translate to standard ASCII from old PET TV ROM business program code:

IF $C1<=X<=SDA THEN X=(X.AND.$7F).OR.S20

Now all characters X> $7B are CBM characters with no ASCII equivalent.

ALGORITHM 1

```
T 1     CMP     #'A
        BCC     DONE
        CMP     #$5B
        BCS     T2
        ORA     #$20
        BNE     DONE
T2 CMP #$CI BCC DONE
        CMP  #$DB  BCS
        DONE AND      #
        $7F

DONL
```

ALGORITHM 2

no code required

ALGORITHM 3

```
T1 CMP #$CI BCC DONE
        CMP  #$DB  BCS
        DONE  AND  #$7F
        ORA      #$20

DONE
```

4. To translate from CBM display code to CBM program code

save bit 6
X=X.AND.$3F
If X<$20 then X=X.OR.$20 If
bit 6=1 then X=X.OR.$80

5. To translate from CBM program code to CBM display code

save bit 7
X=X.AND.$3F
if bit 7 = I then X=X.OR.$40

ALGORITHM 4

```
L1      PHA
        ASL A
        ASL A
        PLA
        PHP
        AND     #$3F
        CMP     #$20
        BCS     L2
        ORA     #$40
L2      PLP
        BCC     DONE
        ORA     #$80

DONE
```

ALGORITHM 5

```
L 1     PHA
        ASL A
        PLA
        AND     #$3F
        BCC     DONE
        ORA     #$40

DONE
```

Appendix F

IEEE Bus Command Character Codes

1. Primary Command Group (PCG) 0-95

A. Addressed command group (ACG) 0-15

1 (GTL) Go To Local
4 (SDC) Selected Device Clear 5 (PPC) Parallel Poll Configure 8 (GET) Group Execute Trigger 9 (TCT) Take Control

B. Universal Command Group (UCG) 16-31

17 (LLO) Local Lockout
20 (DCL) Device Clear
21 (PPU) Parallel Poll Unconfigure 24 (SPE) Serial Poll Enable 25 (SPD) Serial Poll Disable

C. Listen Address Group (LAG) 32-63

32-62 listen address 0-30 63 (UNL) Unlisten

D. Talk Address Group (TAG)

64-94 talk address 0-30
95 untalk

II. Secondary Command Group (SCG) 96-127

Appendix

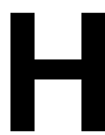
G

System Memory Map

SYSTEM **MEMORY MAP**

0000-00FF	RAM	System page zero variables
0100-01 FF	RAM	6502 hardware stack area
0200-029F	RAM	BASIC input buffer
02AO-03FF	RAM	System absolute variables
0400-7FFF	RAM	32K CBM BASIC text area
0400-3FFF	RAM	16K ('BM BASIC text area
8000-87FF	RAM	CRT display RAM
8800-8FFF		User I/O address space
9000-AFFF		User ROM User I/O
B000-E7 FF	ROM	BASIC Interpreter
E800-EFFF		System I/O
F000-FFFF	ROM	Operating System

Appendix



Assembly Language and System Calls

The purpose in presenting the following list of system calls and variables is to facilitate the interchange of assembly language programs. Programs written in Commodore BASIC have generally been upward compatible between versions 1-4 of BASIC. It is our desire to present a list of assembly language I/O routines that the programmer can use for utilities, interpreters, assemblers and compilers. By using only routines in this list, resulting programs can be I/O independent, and hopefully independent of hardware of future machines. To create new software versions at that time, only a new assembly, with perhaps a different origin, would be required.

Please note that no routines are supported for data structures or mathematics. Both these features are subject to great changes. The user program should handle its own data and communicate with the I/O routines through the standard channels. For math routines, try KIM MATH (available from Commodore in listing form). If this is not suitable, math routines for the 6502 have been published in several journals.

In the following list, functions which are not yet available as a system call, and for which user supplied codes must be included, are flagged with an asterisk.

	ADDRESS	PURPOSE	REGISTERS USED
SYS1	*	set FNLEN, FNADR	A,X,Y
SYS2	*	set LA,FA,SA	A,X,Y
SYS3	F563	open file	none
SYS4	F2E2	close file	A
SYS5	F7AF	open channel for input	X
SYS6	F7FE	open channel for output	X
SYS7	F2A6	clear channel	none
SYS8	F215	input character from channel	A
SYS9	F266	output character to channel	A
SYS10	F205	get buffered character from keyboard	A
SYS11	F2A2	close all files	none
SYS12	*	read disk status, IEEE status	A,X
SYS13	*	set time	A,X,Y
SYS14	*	read time	A,X,Y
SYS 15	F198	talk with attention	A
SYS 16	FOD5	listen with attention	A
SYS17	F1139	unlisten	none
SYS 18	F 1 AE	untalk	none
SYS19	F19E	handshake byte out	A
SYS20	F I CO	handshake byte in	A
SYS21	*	set timeout flag	A
SYS22	F335	check for stop key	none
SYS23	*	scan the keyboard	A
SYS24	*	read top of memory	X,Y
SYS25	*	set top of memory	X,Y
SYS26	F193	secondary address with talk	A
SYS27	F143	secondary address with listen	A
SYS28	*	get address of FAC	X,Y

1. Setting file name length and address.

This subroutine must be called before calling subroutine 3. If a file will be opened without a file name, the file name length must be set to zero. Load the accumulator with the length, X index with the high order address of the file name and Y with the low order address. The file name address can be any valid memory address where the string of characters corresponding to the file name are stored.

```
        LDA #3
        LDX     #>NAME
        LDY     #<NAME
        JSR SYSI

        NAMEBYT `ABC'

        temporary version 4.0 user supplied:
SYS1    STX $DB
        STY     $DA
        STA  $  D9
        RTS
```

2. Setting logical file number, device address, and command.

The logical file number is used as a key by the system to access data stored in a table by the open file subroutine. The device address ranges from 0 to 30 and corresponds to the following CBM devices:

0 Keyboard
1 Cassette #1 2
Cassette #2 3 CRT
display 4 IEEE
printer
8 CBM IEEE disk drive

Device numbers 4 or greater correspond to devices on the IEEE bus.
This subroutine must be called before calling subroutine 3. Load the accumulator with the logical file number, X index with the device number, and Y index with the command. The command is sent as a secondary address on the IEEE following the device number during an attention sequence. If the programmer desires no secondary address to be sent, load Y index with a 255.

Example: logical file #2
device #4 no
command

```
LDA #2
LDX #4
LDY #255
JSR SYS2
```

```
        temporary version 4.0 user supplied:
SYS2    STA $D2
        STX  $  D4
        STY     $D3
        RTS
```

3. Opening a tile.

No registers need to be set up for this routine. However, both subroutines I and 2 must have been called previously. Consider the following example BASIC statement and its implementation in 6502 assembly language:

```
          OPEN 15,8,15,"10"

        LDA   #2
        LDX   #>NAME
        LDY   #<NAME
        JSR   SYSI
        LDA   #15
        LDX   #8
        LDY   #15
        JSR   SYS2
        JSR   $F563

NAMEBYT `10'
```

4. Closing a file.

When all 1/O has completed to a file, call this subroutine with the accumulator loaded with the logical file number used in the open subroutine.

```
        ;CLOSE 15

        LDA #15 JSR
        $F2E2
```

5. Opening a channel for input.

Assuming that a file has been opened by subroutine 3, it can be opened as an input channel. Of course the characteristics of the device will determine if it is valid to do so. This subroutine must be executed before subroutine 8 or 10 is executed for a device other than the keyboard. If input from the keyboard is desired, and there is no association to the logical file number by a previous open file, then the call to this subroutine may be dispensed with.

```
          OPEN LOGICAL FILE 2 AS INPUT CHANNEL

        LDX #2
        JSR $F7AF
```

On the IEEE this subroutine results in sending a talk address followed by a secondary address if one was specified in the open subroutine.

6. Open channel for output.

Assuming that a file has been opened by subroutine 3, it can be opened as an output channel. Of course the characteristics of the device will determine if it is valid to do so. This subroutine must be executed before subroutine 9 is executed for a device other than the CBM CRT. If output to the CRT is desired, and there is no association to an open file by logical file number, then the call to this subroutine may be dispensed with.

OPEN LOGICAL FILE 3 AS OUTPUT CHANNEL

```
LDX #3 JSR $
F7FE
```

On the IEEE this subroutine results in sending a listen address followed by a secondary address if one was specified in the open subroutine.

7. Clear channel.

After opening a channel and performing I/O, this routine closes all open channels and restores the default channels. Input is device 0 and output is 3. This routine may be called optionally by the programmer. An untalk is sent to clear the input channel if the device is on the IEEE. An unlisten is sent to clear the output channel. By not calling this routine and leaving a listener addressed on the IEEE, multiple devices can receive data on the bus. An example would be to address the printer to listen and the disk to talk.

8. Input characters from channel.

A call of this routine will return a character of data from the channel set up by a call to subroutine 5 or the default input channel if no other channel has been set up. Data is returned in the accumulator. The channel remains open after the call. In the case of the keyboard device, the cursor is turned on and continues to blink until carriage return is typed and then characters on the line are returned one by one by calls to this routine. Finally carriage return is sent and the process begins again.

```
JSR $F215
STA DATA
```

9. Output character to channel.

The data to be output is loaded into the accumulator. A call to subroutine 6 sets up the output channel or if this call is omitted, data is sent to the default device which is number 3, the CRT. The character can be transmitted to multiple devices on the IEEE if a clear channel is not performed after the corresponding open channel for output.

```
LDA DATA
JSR $1:266
```

10. Get buffered character from keyboard.

This subroutine removes one character from the keyboard queue and returns an ASCII value in the accumulator. If the queue is empty, the value returned will be zero. Characters are put into the queue by an interrupt driven scan which calls subroutine 23.

```
JSR $F205
STA DATA
```

11. Close all files.

When this subroutine is called, the pointers into the open file table are reset. Additionally, subroutine 7 is called to reset the I/O channels.

```
JSR $F2A2
```

12. Read disk status, IEEE status.

The disk error number is returned in the accumulator and the IEEE status is returned in X index. The disk is assumed to be device 8.

```
        JSR   SYS12
        STA   DS
        STX   ST

        temporary user supplied

SYS12   LDA   #96
        PHA
        JSR   5D995
        PLA
        TAX
        LDY   #0
        LDA   ($OE)Y
        ASL   A
        ASL   A
        ASL   A
        ASL   A
        INY
        STA   $B4
        LDA   ($OE)Y
        AND   #$F
        ORA   $134
        RTS
```

13. Set time.

A system clock is maintained on a 1 /60th second interrupt basis. Three bytes are provided to count jiffies up to 5,184,000---24 hours-at which point the clock rolls over to zero. To set the clock, load accumulator with the most significant, X index with the next most significant, and Y index with the least significant byte of time in jiffies.

```
        ;I0 MINUTES = 36000 JIFFIES

        LDA   #0          ; MOST SIGNIFICANT
        LDX   #>3600
        LDY   #<3600     ;LEAST SIGNIFICANT
        JSR   SYS13

        temporary user supplied
SYS13 SEI
        STA    $80
        STX    $8E
        STY    $8F
        CLI
        RTS
```

14. Read time.

The system clock can be read at any time. Three bytes are returned and their value is 60th's of a second. Accumulator contains most signficant, X index next most significant and Y index the least significant.

```
JSR SYS14 STY TIME STX TIME+1 STA TIME+

        TIME *=*+3



        temporary user supplied
SYS14 SEI
        LDY    $80
        LDX    $8E
        LDA    $8F
        CLI
        RTS
```

15. Talk with attention.

The accumulator is loaded with a device number between 0 and 30. This subroutine ORs in bits to convert this device number to a talk address and then transmits this data as a command on the IEEE bus.

```
;COMMAND DEVICE #4 TO TALK
LDA #4
JSR $F198
```

16. Listen with attention.

The accumulator is loaded with a device number between 0 and 30. This subroutine ORs in bits to convert this device number to a listen address and then transmits this data as a command on the IEEE bus.

```
;COMMAND DEVICE #8 TO LISTEN
LDA #8
JSR $FOD5
```

17. Unlisten.

Use of this subroutine results in an unlisten command being transmitted on the IEEE bus.

```
JSR $F1B9
```

18. Untalk.

Use of this subroutine results in an untalk command being transmitted on the IEEE bus.

```
JSR $F1AE
```

19. Handshake byte out.

The accumulator is loaded with a byte to handshake as data on the IEEE bus. A device must be listening or ST will reflect a timeout.

```
LDA DATA

JSR $F19E
```

20. Handshake byte in.

This routine handshakes a byte of data off the IEEE bus. The data is returned in the accumulator. A device must have been told to talk by subroutine 15.

```
JSR $F1C0
STA DATA
```

21. Set timeout flag.

When the accumulator contains a 0 in bit 7, timeouts are enabled by this routine. A I in bit 7 disables timeouts. Timeouts are a way that the CBM can poll an IEEE device for data without hanging in a handshake sequence. The device must respond to DAV within ()4 milliseconds.

```
        ;DISABLE TIMEOUT
        LDA #0
        JSR SYS21

        temporary user supplied:
  SYS21 STA $03FC RTS
```

'') Check for stop key.

This routine sets the Z flag if the STOP key on the keyboard is pressed while the routine is called. All other flags are maintained.

```
        JSR $F335 BNE
        *+5 JMP
        READY
```

23. Scan the keyboard.

This is the same subroutine called by the interrupt handler. If a key is down, its ASCII value is placed in the keyboard queue.

```
  GET JSR SYS23 ;SCAN KEYBOARD JSR $
        FFDC ;GET CHARACTER CMP #0;
        IS IT NULL?
        BEQ GET      ;YES ...SCAN AGAIN
        STA DATA ;SAVE IT
        JSR $F266    ;PRINT IT


        temporary user supplied:
  SYS23 PHP
        PHA
        TXA
        PHA
        TYA
        PHA
        SEI
        JMP $E455
```

24. Read top of memory.

On power on into BASIC, tests are made to determine the highest RAM memory address. This subroutine returns the hi byte of that address in X index and the lo byte of that address in Y index.

```
        JSR SYS24 STX
        POINT+1 STY
        POINT

        temporary user supplied:
SYS24   LDX   $35
        LDY   $34
        RTS
```

25. Set top of memory.

A programmer may need space for variables to use when BASIC is running. There are 16 bytes available in both pages 0 and 2 but sometimes you may need more. After calling subroutine 24 you can determine where the top of memory is and by loading X and Y and calling this subroutine, you can make a space at the top of RAM for your data.

```
        ;MAKE TOP OF MEMORY $3000
        LDX #>$3000
        LDY #<$3000
        JSR SYS25

        temporary user supplied
SYS25   STX   $35
        STY   $34
        RTS
```

26. Secondary address for talk.

By loading the accumulator with a number between 0 and 31, the user sends a secondary address command over the IEEE with this subroutine. This routine can only be called after subroutine #15.

```
        ;DEVICE #4,SA #5
        LDA   #4
        JSR   $F198     ;SEND TALK ADDRESS
        LDA   #5
        JSR   $F193     ;SEND SECONDARY ADDRESS
```

27. Secondary address for listen.

This routine is essentially the same as subroutine 26 except that it is designed for call after subroutine 16 in a listen sequence.

```
;DEVICE #8 WITH ST #15
LDA #8
JSR $FOD5
LDA #15 JSR
$F143
```

28. Get address of floating accumulator.

In a USR function call, the contents of the parameter expression are evaluated and placed in the floating accumulator. For a detailed description of number format, see the section following the convert format subroutine. This subroutine returns the high order byte of the address of the FAC in X and the low order byte of the address in Y.

```
        JSR SYS28 STX
        POINT+1 STY
        POINT

        temporary user supplied
SYS28   LDX     #>$5E
        LDY     #<$5E
        RTS
```

SYS COMMAND

When it is necessary to transfer control to the machine language program, there are two ways to do it. The preferred approach is the SYS command which transfers control totally from BASIC until control is returned by means of a return from subroutine instruction. It can be used to transfer control to any other program. If the following code is encountered:

```
10 SYS (634)
```

at line 10 BASIC will hand control of the computer to the program located at 634. The general format for the SYS command is:

```
SYS (start address)
```

The start address can be a computed value. In either case, it must result in a positive number not greater than 65535.

CAUTION: Execution of machine language code removes almost all protection that is built into the ROMs to allow the BASIC interpreter to continue functioning without regard to user error. As soon as you transfer control from BASIC' to your own program, any mistakes which occur in your program may cause the machine to cease to function. In order to help solve this type of problem, you should not use the machine language monitor to develop anything other than the most trivial amount of code. In any case, when control of system is lost, it can be regained by repowering the system on.
In order to return from the SYS command, the last instruction in the program which is executed should be a RTS instruction. BASIC will then start interpreting the next statement after the SYS command. In order to pass the variables of data back and forth between the user program and BASIC using the SYS command, data has to be POKEd into temporarily undisturbed memory locations during the execution of the BASIC routine. The results of the SYS operation would have to be PEEKed back into the program that follows the call to SYS.

USR FUNCTION

There are some programs, particularly mathematical ones, in which it is easier to pass parameters to/from BASIC" using the USR function and to get the results directly processed in BASIC. USR is specified with a parameter. BASIC evaluates the expression for its parameter and leaves the results of the evaluation in a floating accumulator which BASIC" uses for all of its functions. It is noted that if no parameter is passed, the floating accumulator is not initializeable by the user or by any other techniques as it is used by BASIC in a variety of ways prior to executing the USR function.
USR calls a routine which executes a machine language program, and leaves a result in the floating accumulator to be analyzed by the BASIC expression. Because USR is a function, it is possible to include the function called user as part of a BASIC instruction as in: IF USR (A)=1, THEN etc. In this case the parameter A will be passed to the USR function in the floating accumulator. The resulting floating accumulator, when the user returns to BASIC, would be compared to 1 and the logical function would be executed.
The SYS command is more useful for transferring control for machine language processing in which variables are not being acted on. USR is more useful when one is trying to implement a new BASIC command. This is an important consideration in using USR. USR uses preassigned variable locations: locations 1 and 2. These locations must be initialized with the hexadecimal value of the starting address in which the machine language program is stored. This can be done anywhere throughout the program with a POKE of the decimal equivalent of the lower address to location I and POKE of the high order address in location 2. Example:

```
10 POKE 1,122
20 POKE 2,2
30 IF USR (A)=1 THEN etc.
```

The parameter specified in the USR function is evaluated, converted to a floating point equivalent with signs, exponent, and mantissa, and placed in a series of bytes which we will call the floating accumulator.

Binary	Integer
1 sign and exponent	4 low byte
2 mantissa **MSB**	5 high byte 3

mantissa
4 mantissa
5 mantissa
6 mantissa LSB
7 sign of mantissa

Binary representation

The exponent is computed such that the mantissa >=1. It is stored as a signed 8 bit binary + $80. Negative exponents are not stored 2's complement. Maximum exponent is 10E38. Minimum exponent is 10E-39 which is stored as $00. A zero exponent is used to flag the number as zero.

Exponent Approximate Value
FF 10E38 A2 10E10 7F 10E-
1 02 10E-38 00 10E-39

Since the exponent is really a power of 2, it should best be described as the number of left shifts (EXP>$80) or right shifts (EXP<=$80) to be performed on the normalized mantissa to create the actual binary representation of the value.
Since the mantissa is always normalized, the high order bit of the most significant byte is always set. This guarantees always at least 40 bits precision which is roughly equivalent to 9 significant digits plus a few bits for rounding. If a number has a value of zero, it may not always have zero bytes in the mantissa. The only true flag for a zero number is the exponent.
If the mantissa is positive, then the sign byte is zero: $00. A negative mantissa causes this byte to be -1: $FF.

Example Floating Point Numbers

1E38	FF	96	76	99	52	00
4E10	A4	95	02	F9	00	00
2E10	A3	95	02	F9	00	00
lE10	A2	95	02	F9	00	00
1	81	80	00	00	00	00
.5	80	80	00	00	00	00
.25	7F	80	00	00	00	00
IE-4	73	D1	B7	59	59	00
1E-37	06	88	1C	14	14	00
IE-38	02	D	C7	EE	EE	00
IE-39	00	A0	00	00	00	00
0	00	00	00	00	00	
-1	81	80	00	00	00	F
-10	84	AO	00	00	00	FF
	(1)	(2)	(3)	(4)	(5)	(6)

Column I - exponent Column 2 - mantissa Column 3 - mantissa Column 4 mantissa Column 5 - mantissa Column 6 Sign of mantissa

Actual floating point BASIC variables are stored in 5 bytes, rather than 6 bytes as is the floating accumulator. Upon examination, one will note that the most signficant byte of the mantissa is always set. If we always assure the number will be in this format, we can use that bit to indicate the sign of the mantissa---thus freeing the byte used for sign. The sixth byte is used in the floating accumulator to simplify operations when shifting the mantissa.

Appendix

I

Commodore BASIC Disk I/O

1.1 INTRODUCTION

User defined parameters will be specified with each command in the next section, but a few general comments can be made here. Parameters are not order dependent, e.g., the drive, file name, and unit could work written in all 6 different permutations of a DLOAD command. A file name may be in quotes, or represented as a string variable enclosed in parentheses. Drive numbers are specified by the letter d followed by a 0 or a 1. The default drive number is generally 0. For commands such as COPY, where the user specifies more than one file name, the default drive for the second filename is the last user specified drive number. The unit# (IEEE address of the disk), is optional on nearly all commands. Unit has a default value of 8. A user specified unit is the letter "U" followed by an integer between 4 and 3 1. "ON Uz" may be written as ",Uz".

The second cassette may not be used simultaneously with the CBM disk commands in version 4.

Disk status variables DS, and DS$, cannot be assigned a value by the user, but when either is referenced as through a PRINT or the right hand side of an expression, the disk command channel will be queried and updated values assigned. The OS keeps a flag to prevent rereading these values if a subsequent disk operation has not been performed.
NOTE: Variables or expressions to evaluate in parameter lists must be enclosed in parentheses, e.g., U(2+B).

1.2 PROGRAM FILE COMMANDS

The following commands are used in manipulation of BASIC' program files:

DSAVE
DLOAD
SCRATCH
RENAME
DIRECTORY

DSAVE "<name>" [,Dx] [ON Uy]

Writes to disk the program that is currently residing in memory.

DLOAD "<name>" [,Dx] [ON Uy]

Loads the program from disk into memory. DLOAD always deletes the current program in memory. In direct mode all files currently open are closed. In program mode, programs can be chained or loaded in sections with access to the same data files. However, the largest program must be loaded first.

SCRATCH "<name>" [,Dx] [ON Uy]

Deletes the file from the disk. File may be program, sequential, or random. An "ARE YOU SURE?" message is given in direct mode to which the user must respond "YES" or "Y" followed by <carriage return> for the command to execute.

RENAME "<namex>" [,Dx] TO "<namey>" [ON Uy]

Change the name of a file. An example use might be to create a backup file before saving a new program of the same name.

DIRECTORY [Dx] [ON Uy]

Display both directories if no drive number is specified. Displays the directory to the screen unless the current output channel directs data to another device.

Disk Data Files
- Sequential and Random

There are two types of disk data files that may be created and accessed by a Commodore BASIC program: sequential files and random access files.

1.3 SEQUENTIAL FILES

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

DOPEN DCLOSE ST DS DS$

PRINT# INPUT# GET#

1.3.1 Creating a Sequential File

he following program steps are required to create a sequential file and access the data in the file:

1. Open the file. You must specify W for Write.	DOPEN#1,"DATA",W
2. Write data to the file using the PRINT# statement.	PRINT#1,A$;B$;C$
3. To access the data in the file, you must CLOSE the file and reOPEN it.	DCLOSE#1 DOPEN#1,"DATA"
4. Use the INPUT# statement to read data from the sequential file into the program.	INPUT#1,X$,Y$,Z$
5. End of file is tested by checking the variable ST.	IF(ST) AND 64 THEN DCLOSE#1

1.3.2 Adding **Data** To A Sequential File

If you have a sequential file residing on disk and later want to add more data to the end of it, you simply open with append, and write the data. Data cannot be inserted however. To do this the entire file must be rewritten.

APPEND#1, "<name>" [,Dx] [ON Uz]

1.4 RELATIVE ACCESS FILES

Relative files are direct access that allow the programmer to position to any record relative to the beginning of the file. Record sizes are fixed in length and may range from I to 254 in size. Record numbers are limited to the capacity of the disk but may not exceed 65535 in number.

The two main components of a relative file are the side sector chain of blocks and the data block chain. Both of these components are linked together through forward pointers similar in fashion to a sequential file.

The side sectors include data block pointers which allow the DOS to move from one record to another within two disk reads. The side sector also contains a table of pointers that point to all of the other side sectors within the file. In order to move from one side sector to the other, the pointer is referenced and the proper track and sector is read. Once the proper side sector is read into memory, the proper data block pointer is referenced, and its track and sector is used to read in the actual data block that the record is contained in.

A file may contain up to six side sectors in it. Each side sector represents 120 data blocks. Therefore, there could be 182,880 bytes within a relative file, assuming a large enough disk capacity. The side sectors do not actually contain any record information, but simply contain information corresponding to the data blocks. The record size dictates where the pointer is placed when a record number is referenced. This is a calculated number when the record number is given through the RECORD command.

When a relative record file is opened for the first time, the DOS will generate one data block and one side sector. If the record size, for example, is 10, then 25 records will be generated automatically upon the opening. To expand the file, simply reference the last record number you wish to generate through the RECORD command and PRINT to that particular record. The intermediate records from the point of the current end of the file to the reference record number, will be generated by the DOS. The generated records will contain a "pi" (CHR$(255)) . Any side sectors and all data blocks necessary to contain a file of this size are also generated.

For example, if the current size of the relative record is one data block long and the record number referenced would expand it to 125 blocks, then an additional side sector would be generated, since one side sector can only represent 120 data blocks.

When a RECORD command positions to a record beyond the current end-of-file, an error #50, RECORD NOT PRESENT, will be generated.

When a channel is opened to a previously existing file, the DOS will simply position to the first record, assuming that the parameters given match properly. The length variable is not necessary on the open if the file is already in existence, but a check is made against the record size that was given in the creating open statement. If the size of the record being written does not match, then error #51, OVERFLOW IN RECORD, will be generated. If a record is smaller than the originally specified length, the remaining bytes will contain nulls. Care should be taken not to position to a null byte within a record, since an attempt to GET from a null byte will cause an endless loop.

The relative channel requires three memory buffers from the system, whereas sequential tiles only require two. Since there are twelve in the system and two of these are used in directory searches and in internal functions, only three relative channels can be open at one time. Another combination might be two sequential channels and two relative channels. Total number of channels open is dictated by the number of buffers used by all of the channels. The highest number of buffers that can be used is ten.

RECORD POSITIONING & INPUT AND OUTPUT TO AND FROM RECORDS

Every time the DOS positions to a record, either through the RECORD command or a previous I/O statement to a record, it performs a read ahead function. For example, if record three was referenced through the RECORD command, then the data block following that which holds record three will be read for further sequential action. This is transparent to the user, since the disk drive performs all 1/O functions independent of the main CPU.

Now in the case just mentioned, if record three was on a boundary between two data blocks, starting in one data block and finishing in another, then the DOS would simply be reading in the remaining part of the record, as well as any following records in the second data block. The records of most relative files will span across data blocks. The only exceptions are record size, 1, 2, 127, and 254. These divide evenly into the 254 size of the data block. Hence, there is no spanning.

This method of spanning gives the efficiency of no overhead, aside from the side sector blocks in the relative files. When a record is written into through the PRINT# statement, the data block is not immediately written out. It is only written out when the DOS moves beyond the particular data block that the record resides in. This can occur through successive printing to sequential records, or positioning to another record outside of that par ticular data block.

This feature reduces the number of disk reads and writes and will improve time efficiency. However, because of this feature it is recommended that two channels not be open to a relative file at one time if either channel will be writing to the relative file. An update may be made in the channel's particular memory buffer area, but the change may not he made on disk until the DOS moves from that particular data block. There is no restriction on this however, and in certain cases where the file is simply read from, it may be advantageous to have more than one channel open to a single relative file.

The relative file has also been designed so that the programmer need not specify record number when printing to it. The relative file can be treated as a sequential file of certain record lengths. Whenever a write or read operation through an input or print statement is performed, the next reference record will be the sequential record. For example, after reading record ten, record eleven will be available for reading or writing. This feature can be used to simulate variable record size in a relative file by using a small record length such as ten and referring to the beginning of a variable record and N number of records thereafter. Each of these groups could be referred to as a logical record.

The DOS terminates printing to a record by sending the EOI signal from the IEEE. This signal is generated on every PRINT# statement, If the PRINT statement goes over the record size a 51 error will be generated, which is record overflow. The information is truncated to the number of characters specified by the record size and the DOS will position to the next record in sequence. If the print statement contains less numbers than the record size, the remaining positions within that record will be filled with nulls. Consequently, when positioning to a record for input the EOI signal is generated from the DOS to the CBM when the last non-zero is transmitted.

If the programmer needs to store binary information a record terminator such as carriage return will have to be used and the record size should be increased by one to accommodate for the terminator.

When generating new data blocks for relative files the requested record number is checked against the number of data blocks left on the diskette. If the resulting number of data blocks is greater than what is left on the diskette, then error #5? is generated file too large.

1.4.1 Creating A Relative File

The following steps are required to create a relative file:

1 . OPEN the file for relative access. Write is implied by the logical record length:

DOPEN#1, "RELATIVE" L20 2.

Write data to each record using PRINT#:

PRINT#1 ,AS

1.4.2 Accessing A Relative File

The following steps are required to update and access relative records:

1. OPEN the file. Read is implied. Record length is carried over from original DOPEN statement

DOPEN#1, "RELATIVE"

2. Use the RECORD command to select the record.

RECORD#1.100

3. INPUT or GET the data:

INPUT#1,A$

1.5 FILE MANIPULATION COMMANDS

HEADER
COLLECT
BACKUP
CONCAT
COPY

HEADER "<name>", Dx [,Izz] [ON Uy]

When Izz, a disk ID number, is specified this command formats the disk specified. The zz may not be a variable. Otherwise, the directory is cleared and the new name assigned to the disk. This command requires caution in this use.

In direct mode, the O.S. prints the following prompt and begins to flash the cursor:

ARE YOU SURE?

The command will not execute without a "YES" or "Y" <RETURN> response from the user. There exists the possibility of a media error with the HEADER command. This can be caused by a missing disk, write protect tab in place, or bad disk surface. The HEADER command will read the command channel and conditionally break with the following error:

BAD DISK

DS$ will contain the actual error message after a ?BAD DISK error.

COLLECT [Dx] [ON Uy]

Frees up space allocated to improperly closed files on the disk and deletes their references from the directory. Follows the chain of file-links through remaining files, re-creating the bit allocation map stored on the disk. Space allocated by the BLOCK-ALLOCATE command is freed.

BACKUP Dx TO Dy [ON Uz]

Duplicate disk x to disk y on unit z complete with disk name, ID, file layout and contents.

COPY [Dx,] "<namex>" TO [Dy,] "<namey>" [ON Uz]

The copy command will only function on drives within a single unit. It can create a copy of a file within the same disk or on the other disk in the unit. Multiple disk units may exist on a PET system.

COPY Dx to Dy [ON Uz]

COPY without file names copies all files from Dx to Dy without altering files that already exist on Dy.

CONCAT [Dx, J "<namex>" TO [DY,] "<namey>" [ON Uz J

Concatenate file namex to file namey.

Appendix J

Index

Note: 1. entries in all caps are statements
2. entries in brackets are key types
3. entries in upper and lower case are general topics

commodore

Commodore Business Machines, Inc
3330 Scott Boulevard
Santa Clara, California 95051

Part Number 321604
Printed in U S A